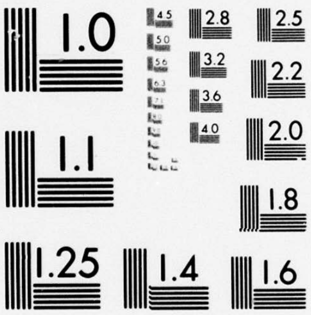
AD-A036 147 NAVAL UNDERWATER SYSTEMS CENTER NEWPORT R I F/G 9/2
OPTIMIZATION TECHNIQUES FOR THE NUSC FORTRAN CROSS-COMPILER.(U)
JAN 77 R A CONVERSE
UNCLASSIFIED NUSC-TR-5214 NL

| OF |
AD
A036147

END
DATE
FILMED
3-77


1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
1.1
1.25
1.4
1.6

NUSC Technical Report 5214

12

Optimization Techniques for the NUSC FORTRAN Cross-Compiler

Robert A. Converse

Combat Systems Control Department

20 January 1977

NAVAL UNDERWATER SYSTEMS CENTER
Newport Laboratory

PREFACE

This research was performed under NUSC Project No. 635P11, Principal Investigator — Robert A. Converse, (Code 3521).

The Technical Reviewer for this report was Thomas P. Conrad, (Code 3521).

REVIEWED AND APPROVED: 20 January 1977

[illegible]
E. L. Messere
Head, Combat Systems Control Department

The author of this report is located at the Newport Laboratory, Naval Underwater Systems Center, Newport, Rhode Island 02840

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR 5214	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER (9) Technical rept.
4. TITLE (and Subtitle) (6) OPTIMIZATION TECHNIQUES FOR THE NUSC FORTRAN CROSS—COMPILER.		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Robert A./Converse		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Underwater Systems Center Newport, Laboratory Newport, Rhode Island 02840		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE (11) 20 January 1977
		13. NUMBER OF PAGES 31
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) (12) 31 p.		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) (14) NUSC-TR-5214 Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) FORTRAN Cross—compiler Code optimization		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) A high-level language permits a programmer to communicate easily with a computer. The machine program that results from the use of a high-level language generally executes slower and requires more of the computer resources than the same program written efficiently in machine language. An optimization pass for the language processor can reduce the execution time and the resource requirements of the resulting program. A machine independent optimization pass that accomplishes such improvements is described in this report. The → over		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

406 088

algorithms that make up the optimization pass include both machine independent general algorithms and a generalization of some machine specific features. The language processor for which optimization is performed is a cross-compiler. Examples are given illustrating the results of the optimization algorithms.

TABLE OF CONTENTS

OPTIMIZATION TECHNIQUES for the NUSC FORTRAN CROSS-COMPILER

INTRODUCTION

A compiler is a computer program that translates statements written in a high-level programming language into a sequence of instructions for subsequent machine execution. A high-level programming language consists of a set of key words and/or symbols combined into a sequence of statements in a manner specified by the rules of syntax for the language. The key words and symbols are generally derived from a notation oriented towards solving a specific class of problems. A programmer defines the procedure that provides a solution to his problem by writing a sequence of statements in the high-level language. An example of a high-level compiler language is FORTRAN. The acronym stands for FORmula TRANslator and the language is best used for scientific applications. A business programmer is more likely to use the Report Program Generator (RPG) language or COBOL, the COmmon Business Oriented Language.

The compiler translates the sequence of high-level language statements into a sequence of machine instructions that actually accomplish what the programmer intends. Ideally, these machine instructions perform the task in the most efficient manner. The determination of which machine instructions are used is made entirely by the compiler. In fact, the programmer does not have to concern himself with the particulars of the machine on which his program executes.

COMPILER COMPONENTS

Compilers are composed of at least three unique parts: the lexical scan, the syntactic analysis, and the code generator. The lexical scan is by far the simplest part of the compiler. It scans the characters of the actual source statement from left to right and extracts the individual key words and symbols (tokens). These tokens are passed to the syntactic analyzer in some internal form. The only tokens passed are those requiring translation, i.e., comments and blanks are eliminated.

The syntactic analyzer disassembles the source program into its basic parts. It completely checks the correctness of the syntax of the source program and produces an internal form of the program that represents the program logic. A principal aspect of this transformation is the construction of tables, such as the symbol table.

The syntactic analyzer must produce an internal form that is a correct representation of a program and that can be efficiently manipulated by subsequent parts of the compiler. There are many possible structures for the internal form. Whatever structure is selected, it must present the program to the next part of the compiler in a correct and usual manner.

The symbol table is an important component of any internal form. It is a collection of all program variables and constants along with their assigned attributes. A reference to a specific variable or constant appearing in the internal form is represented as an index into the symbol table. Thus, all symbol references are represented in a uniform manner to the compiler.

The generation of the internal form of the program is a machine independent operation; however, the lexical scan and the syntax analysis are language dependent, i.e., they are governed by rules of syntax for the specific language. The code generator is just the opposite; it is language independent but machine dependent. The code generation pass requires only the internal form. It processes the internal form and produces a "final", machine dependent, program representation for execution by the target computer. This final representation is called the "object program".

COMPILER STRUCTURES

All compilers perform lexical analysis, syntactic analysis, and code generation. They differ in the manner in which they perform these functions. In some compilers, each individual source statement is processed completely, from lexical scan through code generation, before the next statement is processed. Such compilers are called one-pass compilers. In other compilers, called multi-pass compilers, there are at least two distinct passes: first, the syntactic analyzer builds the internal form for the entire source program, and, second, the internal form of the entire program is passed on to the code generator. In fact, once the code generator is processing the program, the syntactic analyzer can be completely discarded.

OPTIMIZATION

With a multi-pass compiler, passes can be added to the compiler to perform specific functions on the program before any object code is generated. A function that can be performed in this manner is program optimization. The internal form is modified to cause better object code generation than is generat-in the simple two-pass compiler. In general, it is impossible to generate truly optimal code; therefore, a more accurate name for optimization might be code improver with an objective of "code modification in the hope of improvement."[1]

Although optimal code cannot be produced in the "general" case, program performance can be significantly improved by certain modifications to the code generated for specific algorithms. For example, consider the FORTRAN statements

```
A = C + D

B = C + D
```

There is no need to compute the value of the expression C + D twice. However, a non-optimizing compiler would generate code to do just that. Another candidate for code optimization is shown by the FORTRAN statements

```
      IF (A+B) 30, 20, 10

   10 CONTINUE
```

A non-optimizing compiler would generate the following

```
      "EVALUATE (A+B), CALL IT X"

      "JUMP TO STATEMENT 30 IF X NEGATIVE"

      "JUMP TO STATEMENT 20 IF X ZERO"

      "JUMP TO STATEMENT 10 IF X POSITIVE"

   10 CONTINUE
```

Obviously, the "JUMP TO STATEMENT 10" instruction could be completely removed without affecting the program logic.

OPTIMIZATION COSTS

There are trade-offs in terms of time and computer resources associated with each optimization. Generally, the choice is between a costly program compilation that decreases program execution time or a rapid compilation with its less efficient program execution.

A second factor that must be considered is program debugging. If the machine code is modified to produce more efficient code, then the instruction execution sequence may not correspond exactly to the sequence of high-level statements written by the programmer. This can cause considerable confusion when a programmer attempts to isolate and correct errors. An example of this is given by the FORTRAN statement

C = A * 2.

An equivalent form for this expression is

C = A + A.

If the computer can execute an add instruction faster than it can execute a multiply instruction, then a legitimate code optimization is to generate the add A to A sequence in place of the multiply A by 2 sequence. If an error occurs as a result of the add (e.g., "register overflow on ADD"), the programmer could become confused since no add operations appear in his program.

CODE IMPROVEMENT

The objective of the code optimization process is to improve the efficiency of program execution. There are two areas in which code improvement techniques can be applied. One area for improvement is at an intermediate language level, where the syntactic analysis portion of the compilation is complete and no machine code has yet been generated. At that point, optimizations such as those presented in the previous examples can be accomplished. These are called machine independent optimizations.

The second area for code improvement is at the code generation level. The code generator may be able to take advantage of certain machine instructions to accomplish specific tasks. For example, the expression

A = A + C,

generates a LOAD, ADD, STORE sequence for the general case. If C is the constant "1" and the computer has an "Increment Memory" instruction, the code generator could generate one instruction to increment the variable A. Optimizations of this type are machine dependent optimizations and are unique to a language implementation on a specific computer.

Machine independent optimizations can be performed at two levels: local optimization and global optimization. Local optimization is that which can be made at specific points in a program using only information about the immediate program area. Global optimization requires knowledge of an entire program or at least a very large portion of a program.

OBJECTIVE

This research was directed at demonstrating the applicability of some generalized optimization techniques to an existing language. The generality of the technique was maintained by delaying selection of a target computer until the optimization transformations were performed. In addition, some machine dependent optimizations are generalized for application to two quite different computers.

THE LANGUAGE AND THE MACHINES

The FORTRAN language was developed in the early 1960's for algebraic and computational applications. Definition of a national standard in 1966 contributed to widespread acceptance of the FORTRAN language throughout the computer world. NUSC developed the FORTRAN cross-compiler that was used in this study. Hosted by the UNIVAC AN/UYK-7 computer, this compiler is capable of generating object code for either the AN/UYK-7 computer or for the UNIVAC AN/UYK-20 mini-computer. The compiler is a multi-pass compiler. The first pass consists of the lexical scan and the syntax analysis. The final pass generates relocatable object code for subsequent loading and execution. Intermediate passes, when selected at compile time, perform optimization.

INTERNAL FORM

The first pass generates a machine independent internal form of the program. The internal form is made up of a set of 3-tuples (triples). Each triple consists of an operator and two operands, i.e.,

OPERATOR OPERAND 1 OPERAND 2

The OPERATOR is an index into a table of actual operators. OPERAND 1 and OPERAND 2 are pointers to the operands at which OPERATOR is applied. For example the expression,

A + B,

is represented by the triple

(POINTER TO THE PLUS OPERATOR)

(SYMBOL TABLE INDEX FOR VARIABLE A)

(SYMBOL TABLE INDEX FOR VARIABLE B)

A triple always contains an operator. However, it contains only as many operands as are required for the operator. Thus, the statement

GO TO 10

produces the triple

(POINTER TO GO TO OPERATOR)

(SYMBOL TABLE INDEX FOR STATEMENT NUMBER 10)

(NOT USED)

The triples produced by the first pass are machine independent. They are the result of an analysis of the syntactic and semantic content of each statement. The rules of syntax for the language strictly define the grammatical structure for a statement in the language. Associated with the rules of syntax are transformation rules for determining the meaning of the statement (semantic rules). The triples represent the sequence of operations required to accomplish the programmer's objectives as determined by the syntactic and semantic analysis.

The final pass processes the set of triples sequentially to produce relocatable machine code. As stated in the preceding paragraph, the set of triples is machine independent. The actual computer for which code is to be generated is immaterial until the code generator is invoked. To generate object code for any computer, the implementor of the translator only has to produce the code generation pass. The internal form, the machine independent set of triples, is its standard input.

THE HOST MACHINES

The AN/UYK-7 FORTRAN cross-compiler can generate machine code for two computers: the AN/UYK-7 computer and the AN/UYK-20 minicomputer. The AN/UYK-7 computer is a two-state general purpose digital computer. It is modular in design. In its minimum configuration it contains a Central Processing Unit (CPU), 3 memory banks, and an Input/Output Controller (IOC); in its maximum configuration it contains 3 CPU's, 16 memory banks, and 4 IOC's. Each CPU has a unique set of operational registers for each state (executive state or task state). Each set of registers consists of eight arithmetic accumulators, seven index registers, and eight base registers. In addition, the executive state has a set of memory protection registers for memory access control. The CPU instruction set includes several privileged instructions that can be executed only from the executive state. The IOC is started and stopped by privileged CPU instructions. Once started, the IOC has direct access to memory for its input/output operations. Each memory bank contains 16,384 words, and each word contains 32 bits.

The AN/UYK-20 mini-computer is also modular in design. Its minimum configuration consists of a CPU , a memory, and an IOC. The CPU is micro-programmable, with the micro-program specified at time of manufacture. It is a task state computer with 16 general purpose registers. The IOC is started and stopped by the CPU and has direct access to memory. Memory is available in blocks of 8,192 16-bit words to a maximum of 65,536 words.

These two computers have several differences of interest to the compiler writer, as outlined in table 1.

Table 1. Differences Between the AN/UYK-7 and the AN/UYK-20 Computers

AN/UYK-7 COMPUTER	AN/UYK-20 MINI-COMPUTER
32-bit word size	16-bit word size
Performs arithmetic using a 1's complement convention	Performs arithmetic using a 2's complement convention
Has separate general registers (accumulators), index registers, and base registers	Has general purpose registers
Address computation involves a base register	Has a hardware paging capability
Has floating point instructions in its instruction repertoire	Must perform floating point arithmetic interpretively
Has a fixed instruction repertoire	Can have user specified micro-programs included as a part of its instruction repertoire

MACHINE INDEPENDENT OPTIMIZATIONS

Machine independent optimizations are those optimizations that can be performed without considering the specific capabilities of the computer hardware. Machine independent optimization is carried out by modifying the internal form to eliminate redundant computations, eliminate unnecessary statements, and evaluate constant expressions. An essential element in the optimization process is control flow analysis.

CONTROL FLOW ANALYSIS

Control flow analysis is a codification of the logical relationship between statements in a program. There are many methods available to conduct this codification, but the underlying motivation of each is the description of the program execution sequence (its logical flow). The purpose of control flow analysis is to answer questions such as: Is this a loop? Has this expression already been evaluated? Is this expression ever executed? A control flow analysis describes relationships between statements and ensures that the result of optimization is semantically correct, i.e., the program does what the programmer wanted it to do.

The minimum flow analysis required defines the areas in the program to which local machine independent optimization algorithms can be applied. Bagwell[2] shows a machine independent optimization procedure that optimizes a program on a statement by statement basis. For this optimization procedure, the flow analysis is trivial.

BASIC BLOCKS

The initial goal of flow analysis is to partition the program into a set of program units called basic blocks. A basic block is a sequence of statements for which there is one entrance statement, from which there is one exit statement and within which each statement is executed. The examples that follow in table 2 illustrate this concept as applied to FORTRAN statements.

The codification problem is simplified by defining basic blocks. Further flow analysis is accomplished by describing the relationships between basic blocks. For each basic block there is a physical predecessor block and a physical successor block. In table 2, for example, block 2 is the physical predecessor of block 3 and the physical successor of block 1.

Table 2. Basic Blocks

	A = B	
	CALL SUBRI	Block 1
	GO TO 20	
	C = D	Block 2
20	E = SIN(F)	
	G = E + H	Block 3
	IF (H. EQ. O) GO TO 30	
	T = Q	Block 4
30	CONTINUE	
	RETURN	Block 5
	END	

Each basic block also has one or more logical predecessors and one or more logical successors associated with it. The logical predecessors of a basic block are the basic blocks from which control is obtained. The logical successors of a basic block are the basic blocks to which control can be transferred. For example, block 1 in table 2 is the logical predecessor to block 3; conversely, block 3 is the logical successor to block 1.

Note that a subroutine call in block 1 does not constitute a basic block. Similarly, function references, as in block 3, are permissible within basic blocks. These statements are permitted because the flow of control is sequential: the logical and physical predecessor and successor for the subprogram reference within the basic block are the same statements. However, there are special problems with subprogram references in a basic block. These are discussed in more detail in subsequent paragraphs.

The NUSC FORTRAN cross-compiler accomplishes the codification of control flow analysis by assigning a unique index to each basic block. This index is computed by counting the basic blocks as they physically appear in a subprogram and assigning this count as the index. For each basic block, the indices of the logical predecessor and logical successor blocks are saved in a predecessor list and a successor list, respectively. Pointers into these lists are found in a master list that is indexed by the basic block index for which the logical predecessor or logical successor is required.

LOCAL MACHINE INDEPENDENT OPTIMIZATIONS

The NUSC FORTRAN cross-compiler performs local optimizations within a basic block. If it is certain that each statement is executed, it is often possible to eliminate redundant computations. This procedure is known as common subexpression elimination. Each expression is compared to all other expressions in the basic block. If two or more expressions are found to be identical, only the first occurrence of the expression is evaluated. The result is saved, and for subsequent occurrences of the expression, the result is referenced, eliminating the need to reevaluate the expression.

Isolating candidate expressions for elimination poses some interesting problems. The basic problem is recognition of common subexpressions. For example, the FORTRAN expressions A + B and B + A are common subexpressions and are legitimate candidates for optimization. However, because of the different order in which the operands are written, a straightforward lexical comparison will not find the match. To resolve this problem, the properties of the operators must be known. For associative operators, i.e., those operators for which the order of the operands is unimportant, the operands are rearranged into a canonical form such as alphabetical order. In this example, such a rearrangement would allow the common subexpression to be identified. (The canonical form actually used in this optimizer is the ascending order of symbol table index. It is only necessary that a definite ordering of the operands be accomplished.)

Operators that do not have the associative property are candidates for optimization if some transformation can be found to make the operation associative. For example, the expression A-B is transformed into the expression A + (-B). The complement is associated with the operand, (-B), and the actual operator is now associative. Frailey[3] presents a set of algorithms for finding common subexpressions among expressions having unary operators, inverse operators, and complements. Lee[4] presents a recognition process for triples that is faster than the direct comparison approach. This process, first suggested by Gries,[5] uses a "dependency number" system. In this process, only triples with the same dependency number have to be checked for commonality.

For this application, the technique suggested by Lee has been rejected in favor of Frailey's approach to expression optimization. Frailey's approach provides a ready mechanism for identifying and eliminating factors within an expression. This is extremely difficult using Lee's approach. The factor problem is demonstrated in the expressions A + B + C and A + C + D. The common factor between the two expressions is A + C. In using Frailey's algorithms, the common subexpression A + C would be found and its occurrence in the second expression eliminated. If Lee's approach is used, the common factor would not be found because the comparison is made against triples and the triple for the factor A + C does not appear in both expression representations.

There are some areas of concern in optimization within basic blocks that are not important for single statement optimization. Chief among them is the integrity of the variables included in the common subexpression. Even though two expressions may be identical, if the value of one of the variables has changed since the first occurrence of the expression, then the two expressions are in fact different and the second expression must not be eliminated. A further complicating factor is the integrity of COMMON elements, or actual arguments passed to a subroutine or function. To ensure the general validity of COMMON elements and arguments, candidates for elimination preceded by a subprogram reference are not eliminated if an operand is an element in COMMON or is an actual argument for that subprogram reference. For similar reasons, a variable that has been included in an EQUIVALENCE statement is not a candidate for optimization across statement boundaries.

The machine independent optimizations suggested by Bagwell[2] have been applied to basic blocks as well. In particular, constant expressions are evaluated. Thus, the expression A = 3.14159*2.0 becomes A = 6.28318 after constant expression evaluation. Note that this requires the definition of a new constant. Similarly, the computational complexity of an expression is reduced. For example, the expression A**2 is changed to A*A. For most computers, this transformation produces machine code that executes faster.

Optimizations within a basic block are performed in a specific order. The NUSC FORTRAN cross-compiler performs them in the following order:

1. Evaluate constant expressions

2. Reduce expression complexity

3. Perform common subexpression elimination

4. Define new operators.

Optimizations 1 and 2 are performed using the internal form produced in the first pass. The result of these optimizations is the deletion of some triples and the modification of others. In particular, the triples that define the constant expression are deleted and the resulting constant is used. Triples that represent multiplication or exponentiation operators are examined and changed, if possible, to addition or multiplication, respectively, with a corresponding change to the operands.

The third optimization, common subexpression elimination, is the implementation of Frailey's algorithms. This process consists of the following steps.

1. Examine each operator. If necessary, transform the operator so that it is associative or associative and commutative. The transformation

process produces an addition operator from a subtraction operator and a multiplication operator from a division operator. The negative or inverse property is associated with the appropriate operand.

2. Define N-tuples. Modify the internal form so that for each operator there are N operands, $(N > 1)$. For example, the triples produced by the first pass for the expression $A + B + C$ are:

```
(PLUS)
          (POINTER TO SYMBOL A)
          (POINTER TO SYMBOL B)
(PLUS)
          (POINTER TO PREVIOUS TRIPLE)
          (POINTER TO SYMBOL C)
```

the transformation to N—tuple notation produces:

```
(PLUS)
          (POINTER TO SYMBOL A)
          (POINTER TO SYMBOL B)
          (POINTER TO SYMBOL C)
```

3. Arrange the operands in canonical form. Because the operator is associative and commutative, the order in which the operator is applied to the operands is not significant.

4. Search for common subexpressions. The arrangement of operands in canonical form simplifies the search procedure and allows identification of factors without checking all possible combinations of operands.

5. Replace subsequent occurrences of a common subexpression with a pointer to its first appearance.

6. Transform the N—tuples back to triples for processing by subsequent passes.

The fourth optimization scheme examines the internal form for specific code sequences that are candidates for machine dependent optimizations. These sequences are discussed in greater detail in the section entitled "Machine Dependent Optimizations".

GLOBAL MACHINE INDEPENDENT OPTIMIZATIONS

The NUSC FORTRAN cross-compiler performs flow analysis while the local machine independent optimizations are being performed. Analysis of the logical flow isolates those basic blocks for which no logical predecessor exists. Such a block is a candiadate for elimination. If the block is a data block or an entry block for a subprogram, it is retained. Otherwise, it is eliminated since there is no way that control can pass to that block. Redundant jumps are eliminated as a result of examining the logical flow relationship and the physical relationship between blocks. Note that the elimination of a basic block can affect this relationship. For example, processing the FORTRAN sequence:

```
      GO TO 10
      A = B
   10 C = D
```

eventually results in the elimination of the statement GO TO 10. First, the basic block A = B is eliminated and that, in turn, permits the elimination of the GO TO 10.

MACHINE DEPENDENT OPTIMIZATIONS

Machine dependent optimization involves the selection of features that are unique to a specific computer. An example of such a feature is the MOVE CHARACTER instruction that is a part of the IBM 360 and 370 family of computers. The selection of machine instructions is accomplished by the code generation pass of the compiler. The optimizing pass can assist the code generator by indicating some code sequences that are candidates for machine dependent optimization.

There are several statement forms that occur with sufficient frequency to merit special consideration. These include statements such as:

1) A = -B

2) A = A + 1

3) A = A + B

Some computers can accomplish these operations in fewer machine instructions than are required in the general case. Statement 1 causes a general machine sequence to be generated that includes:

```
LOAD B

COMPLEMENT

STORE RESULT IN A.
```

Some computers can perform a sequence such as:

```
LOAD AND COMPLEMENT B

STORE RESULT IN A.
```

This results in a shorter program by one instruction and a faster program by one instruction for each occurrence of a statement of the form A = -B. Similarly, typical code for example 2 is:

```
LOAD A

ADD 1

STORE RESULT IN A.
```

In some machines, this can be replaced with:

```
INCREMENT A IN MEMORY.
```

The optimizer has defined pseudo operators (macros) that isolate these common code sequences for the code generator. As stated previously, these new operators are defined during the first pass of the optimization process. The new operators that have been defined include the following.

1. REPLACE ADD (RADD)

   Process expressions of the form A = A + expr.

2. REPLACE SUBTRACT (RSUB)

   Process expressions of the form A = A -expr.

3. REPLACE INCREMENT (RINC)

   Process expressions of the form A = A + 1.

4. REPLACE DECREMENT (RDEC)

   Process expressions of the form A = A - 1.

5. MULTIPLY BY 2 RAISED TO AN INTEGER POWER

   (Shift left (LSH) or shift right (RSH) as appropriate). For multiplication of an operand by an operand that is an exact power of two, allow the code generation pass to use a shift instruction sequence.

The criterion for defining a pseudo operator is that a straightforward instruction sequence must exist for those computers that do not have a machine instruction to accomplish the pseudo operator directly.

The code generation pass for each target computer must be modified to accept these pseudo operators. For the target computers involved in this project, the change involved writing a new handler for the operator and providing the linkage to that handler. Table 3 shows the results.

Table 3. Sample Machine Sequences

STATEMENT	AN/UYK—7 CODE	AN/UYK—20 CODE
A = A + B	LOAD REGO, B RADD REGO, A	LOAD REGO, B ADD REGO, A STOR REGO, A
A = A + 1	RINC REGO, A	LOAD REGO, 1 ADD REGO, A STOR REGO, A
A = A -B	LOAD REGO, B RSUB REGO, A	LOAD REGO, A SUB REGO, B STOR REGO, A
A = A -1	RDEC REGO, A	LOAD REGO, A SUB REGO, 1 STOR REGO, A
A = A * 4	LOAD REGO, A LSH REGO, 2 STOR REGO, A	LOAD REGO, A LSH REGO, 2 STOR REGO, A
A = B / 8	LOAD REGO, B RSH REGO, 3 STOR REGO, A	LOAD REGO, B RSH REGO, 3 STOR REGO, A

In table 3, A and B are variables and REGO indicates any arithmetic register. Note that for the first four statements, machine dependent instructions were used on the AN/UYK-7 computer whereas a straightforward implementation of the instruction sequence was possible with the AN/UYK-20 mini-computer.

The sequence for the left shift (multiply by a power of two) requires the same number of instructions as for a multiply instruction. The improvement in this case is in the instruction execution speed. Generally, a shift instruction executes faster than a multiply instruction. In the target computers used in this project, the shift instruction executes more than four times faster than the multiply instruction. The code generation handler for the left shift macro operator could be the same handler as the multiply handler for a target computer whose execution speed is the same for both operations.

A divide sequence that uses the right shift operator requires fewer instructions than the standard divide operator because a standard divide instruction requires that the dividend be right justified in a double register. Thus, in order to use a divide instruction, the registers must be initialized. (One initialization procedure is to multiply the dividend by the constant 1.) Use of the right shift instruction eliminates the necessity of preparing for division. Also, for the computers in use for this project, the shift instruction is faster by a factor of more than eight.

RECOMMENDATIONS FOR FURTHER DEVELOPMENT

The optimizer described in this paper generates an internal form of a program that produces more efficient object code than that produced by the non-optimized internal form. There are still other areas in which improvement can be made. The algorithms described operate on arithmetic operators. Similar algorithms can be defined for logical or relational operators. Additional effort can be spent in the analysis of loops and the possibilities for removing loop invariant operations from a loop.

For this project, the target computer was not an important consideration. The input to the optimization pass is an internal form of a program and a symbol table containing its symbolic names and constants. Any language processor capable of producing a compatible internal form and symbol table could be linked to this optimizer and to the set of code generators that are defined. Thus, the optimizer is, or with little effort can be made to be, language independent as well as machine independent.

REFERENCES

1. J. Cocke, "Global Common Subexpression Elimination," ACM SIGPLAN Notices, July 1970, pp. 20-24.

2. J. T. Bagwell Jr., "Local Optimization," ACM SIGPLAN Notices, July 1970, pp. 52-66.

3. D. J. Frailey, A Study of Code Optimization Using a General Purpose Optimizer, University Microfilms, Ann Arber, Mich., 1971.

4. J. A. Lee, The Anatomy of a Compiler, second edition, Van Nostrand Reidhold Co., New York, 1974.

5. D. Gries, Compiler Construction for Digital Computers, John Wiley and Sons, Inc., New York, 1971.

BIBLIOGRAPHY

Aho, A., R. Sethi, and J. D. Ullman, "A Formal Approach to Code Optimization," ACM SIGPLAN Notices, July 1970, pp. 86-100.

Allen, F. E., "Control Flow Analysis," ACM SIGPLAN Notices, July 1970, pp. 1-9.

Busam, V. A. and D. E. Englund, "Optimization of Expressions in FORTRAN," Communications of the ACM, vol. XII, December 1969, pp. 666-674.

Frailey, D. J., "Expression Optimization Using Unary Complement Operaters," ACM SIGPLAN Notices, July 1970, pp. 67-85.

Lowry, D. J., S. Edward, and C. W. Medlock, "Object Code Optimization," Communications of the ACM, vol. XII, January 1969, pp. 13-22.

Nicholls, J. E., The Structure and Design of Programming Languages, Addison-Wesley Publishing Company Inc., Reading, Mass. 1975.

Rustin, R., Design and Optimization of Compilers, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1972.

Wulf, W., R. K. Johnson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, The Design of an Optimizing Compiler, American Elsevier Publishing Co., Inc., New York, New York, 1975.

APPENDIX A

EXAMPLES

For each example that follows, two tables are provided. The first table is the set of triples output by the first pass of the compiler (the syntactic analysis). The second table is the set of triples produced as a result of the optimizations performed on the output of that pass.

For each table, there are at least four fields: an index into the table, a symbol or mnemonic that represents the operator, and two operands. The index progresses in increments of two because each table entry requires two words. The operands are indicated by their symbolic name, the constant value, a pointer to another triple (indicated by the prefix TRI), or blank (indicating that the operand is not used). The meanings of the operators are outlined in table A1.

Table A1. Operator Meanings

OPERATOR	MEANING
+	Add operand 1 to operand 2
-	Subtract operand 2 from operand 1
*	Multiply operand 1 by operand 2
/	Divide operand 1 by operand 2
=	Assign operand 2 to operand 1
**	Raise operand 1 to the operand 2 power
GO	Unconditional branch to operand 1
CGO	Computed branch, operand 1 is the switch index, operand 2 is the maximum number of switch points
GOAR	Argument of switch, operand 1 is the statement label, operand 2 is the index for this label
BRN	Branch to operand 2 if the expression pointed at by operand 1 is negative
BRZ	Branch to operand 2 if the expression pointed at by operand 1 is zero
BRP	Branch to operand 2 if the expression pointed at by operand 1 is positive
RADD	Add operand 1 and operand 2; store the result in operand 1
RSUB	Subtract operand 2 from operand 1; store the result in operand 1
RINC	Add 1 to operand 1; store the result in operand 1
RDEC	Subtract 1 from operand 1; store the result in operand 1
LSH	Shift operand 1 left by operand 2 (multiply operand 1 by 2 to the operand 2 power)
RSH	Shift operand 1 right by operand 2 (divide operand 1 by 2 to the operand 2 power)
END	No operands, end of the subprogram
LINE	Operand 1 is the source statement line number

Example A-1: Constant Expression Evaluation

SOURCE STATEMENTS

```
I = 1 + 2 + 3 * 4
J =4 ** 2
B = 8. / 2.
```

Triples after syntactic analysis

INDEX	OPERATOR	OPERAND 1	OPERAND 2
0	LINE	1	
2	+	1	2
4	*	3	4
6	+	TRI 2	TRI 4
8	=	I	TRI 6
10	LINE	2	
12	**	4	2
14	=	J	TRI 12
16	LINE	3	
18	/	8.	2.
20	=	B	TRI 18

Triples after optimization

INDEX	OPERATOR	OPERAND 1	OPERAND 2
0	LINE	1	
2	=	I	15
4	LINE	2	
6	=	J	16
8	LINE	3	
10	=	B	4.

Example A-2 Common Subexpression Elimination

SOURCE STATEMENTS

```
A = B + C

D = B + C

B = A * D

C = D * A

A = B + C + D

E = D + B

F = B + C + D

G = B + C
```

Triples after syntactic analysis

INDEX	OPERATOR	OPERAND 1	OPERAND 2
0	LINE	1	
2	+	B	C
4	=	A	TRI 2
6	LINE	2	
8	+	B	C
10	=	D	TRI 8
12	LINE	3	
14	*	A	D
16	=	B	TRI 14
18	LINE	4	
20	*	D	A
22	=	C	TRI 20
24	LINE	5	
26	+	B	C
28	+	TRI 26	D
30	=	A	TRI 28
32	LINE	6	
34	+	D	B
36	=	E	TRI 34
38	LINE	7	

40	+	B	C
42	+	TRI 40	D
44	=	F	TRI 42
46	LINE	8	
48	+	B	C
50	=	G	TRI 48

Triples after optimization

INDEX	OPERATOR	OPERAND 1	OPERAND 2
0	LINE	1	
2	+	B	C
4	=	A	TRI 2
6	LINE	2	
8	=	D	TRI 2
10	LINE	3	
12	*	A	D
14	=	B	TRI 12
16	LINE	4	
18	=	C	TRI 12
20	LINE	5	
22	+	B	D
24	+	TRI 22	C
26	=	A	TRI 24
28	LINE	6	
30	=	E	TRI 22
32	LINE	7	
34	=	F	TRI 24
36	LINE	8	
38	+	B	C
40	=	G	TRI 38

Example A-3: New operations

SOURCE STATEMENTS

```
I = I + 1

J = J - 1

A = A + B

C = C - D

L = M * 16

M = N / 8
```

Triples after syntactic analysis

INDEX	OPERATOR	OPERAND 1	OPERAND 2
0	LINE	1	
2	+	I	1
4	=	I	TRI 2
6	LINE	2	
8	-	J	1
10	=	J	TRI 8
12	LINE	3	
14	+	A	B
16	=	A	TRI 14
18	LINE	4	
20	-	C	D
22	=	C	TRI 20
24	LINE	5	
26	*	M	16
28	=	L	TRI 26
30	LINE	6	
32	/	N	8
34	=	M	TRI 32

Triples after optimization

INDEX	OPERATOR	OPERAND 1	OPERAND 2
0	LINE	1	
2	RINC	I	
4	LINE	2	
6	RDEC	J	
8	LINE	3	
10	RADD	A	B
12	LINE	4	
14	RSUB	C	D
16	LINE	5	
18	LSH	M	4
20	=	L	TRI 18
22	LINE	6	
24	RSH	N	3
26	=	M	TRI 24

Example A-4: Unused code removal

SOURCE STATEMENTS

```
                GO TO (10, 20, 30), I
                GO TO 50
                A = B
      10        IF (A - B) 20, 30, 40
                C = D
```

Triples after syntactic analysis

INDEX	OPERATOR	OPERAND 1	OPERAND 2	BLOCK #
0	LINE	1		1
2	CGO	I	3	1
4	GOAR	10	0	1
6	GOAR	20	1	1
8	GOAR	30	2	1
0	LINE	2		2
2	GO	50		2
0	LINE	3		3
2	=	A	B	3
0	LINE	4		4
2	-	A	B	4
4	BRN	TRI 2	20	4
6	BRZ	TRI 2	30	4
8	BRP	TRI 2	40	4
0	LINE	5		5
2	=	C	D	5

Triples after optimization

INDEX	OPERATOR	OPERAND 1	OPERAND 2	BLOCK #
0	LINE	1		1
2	CGO	I	3	1
4	GOAR	10	0	1
6	GOAR	20	1	1
8	GOAR	30	2	1
0	LINE	2		2
2	GO	50		2
0	LINE	3		3
0	LINE	4		4
2	-	A	B	4
4	BRN	TRI 2	20	4
6	BRZ	TRI 2	30	4
8	BRP	TRI 2	40	4
0	LINE	5		5

INITIAL DISTRIBUTION LIST

Addressee	No. of Copies
NRL	1
NAVCOASTSYSLAB	1
NAVSEASYSCOM (SEA-034, -660D, PMS-393)	3
NAVSURFWPNCEN	1
NELC	1
NAVUSEACEN	1
NAVSEC	1
NAVPGSCOL	1
NAVWARCOL	1
APL , U. of Wash.	1
ARL, Penn State	1
APL, Johns Hopkins	1
ARPA	1
DDC, Alexandria, Va	12